Aegaeon: Effective GPU Pooling for Concurrent LLM Serving on the Market

Yuxing Xiang[1], Xue Li[2], Kun Qian[2], Yufan Yang[2],
Diwen Zhu[2], Wenyuan Yu[2], Ennan Zhai[2], Xuanzhe Liu[1], Xin Jin[1], Jingren Zhou[2]

1 北京大學 PEKING UNIVERSITY

2 Alibaba Cloud

Models: growing variety

- Modern model markets feature many different LLMs
  - Thousands of models on Hugging Face
  - Growing faster with fine-tuning services like Tinker

Qwen deepseek LLaMA

Hugging Face Alibaba Cloud Model Studio Tinker

Models: growing variety

- Modern model markets feature many different LLMs
  - Thousands of models on Hugging Face
  - Growing faster with fine-tuning services like Tinker

Qwen deepseek LLaMA

Concurrent LLM serving

Many models at the same time

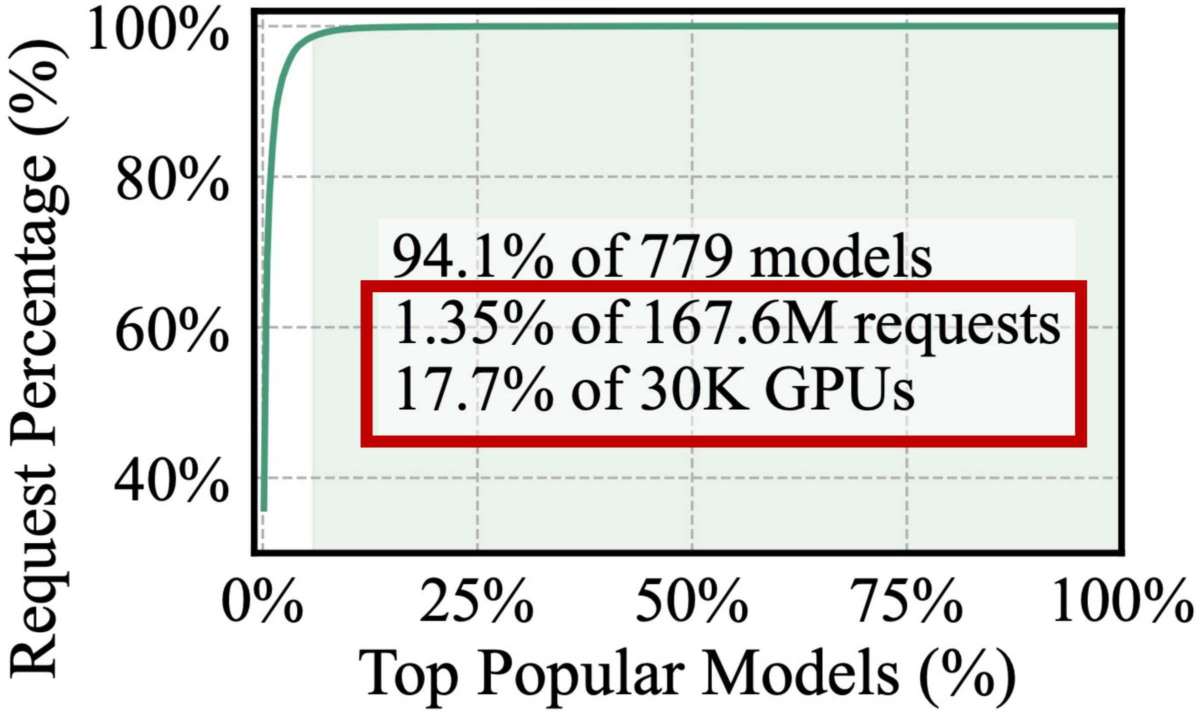
Hugging Face

Alibaba Cloud Model Studio

Tinker

Workload: highly skewed

- ❖ A one-hour concurrent workload from Alibaba Cloud Model Studio
  - Over 90% of models, with less than 2% of requests
  - **Long tail of low-rate models**
  - **Reserving GPUs**: Less than 0.2 RPS per GPU



167.6M x 1.35% req
/ (17.7% x 30K GPU)
/ (3600s)

≈ 0.118 RPS / GPU

Concurrent LLM serving is extremely wasteful!

Concurrent LLM serving
(w/ GPU reservation)
- One model per GPU
- < 0.2 RPS per GPU

Single-LLM serving
- Optimized with SGLang
- > 2 RPS per GPU*

10x Resource Efficiency Gap!

* Estimated with SGLang's DeepSeek deployment blog.

Closing the gap with GPU Pooling

Concurrent LLM serving
(w/ GPU reservation)

- One model per GPU
- < 0.2 RPS per GPU

Single-LLM serving

- Optimized with SGLang
- > 2 RPS per GPU*

Closing the gap with GPU Pooling

Concurrent LLM serving
(w/ GPU reservation)
- One model per GPU
- < 0.2 RPS per GPU

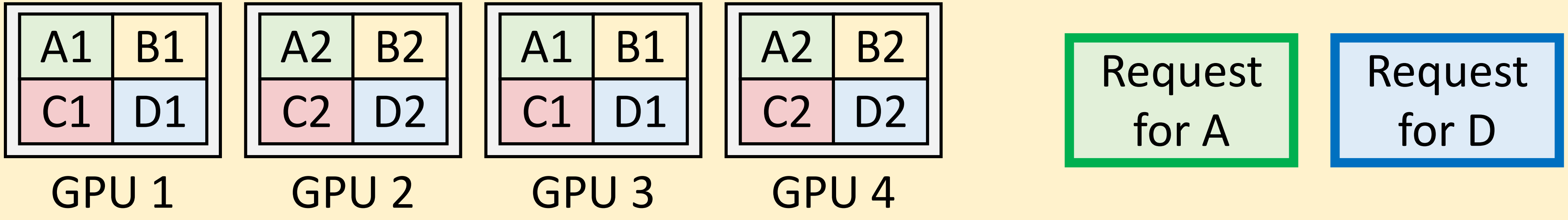
Concurrent LLM serving
(w/ GPU pooling)
- Multiple (n) models per GPU
- 0.2 x n RPS per GPU

Single-LLM serving
- Optimized with SGLang
- > 2 RPS per GPU*

Serve as many models as possible per GPU!

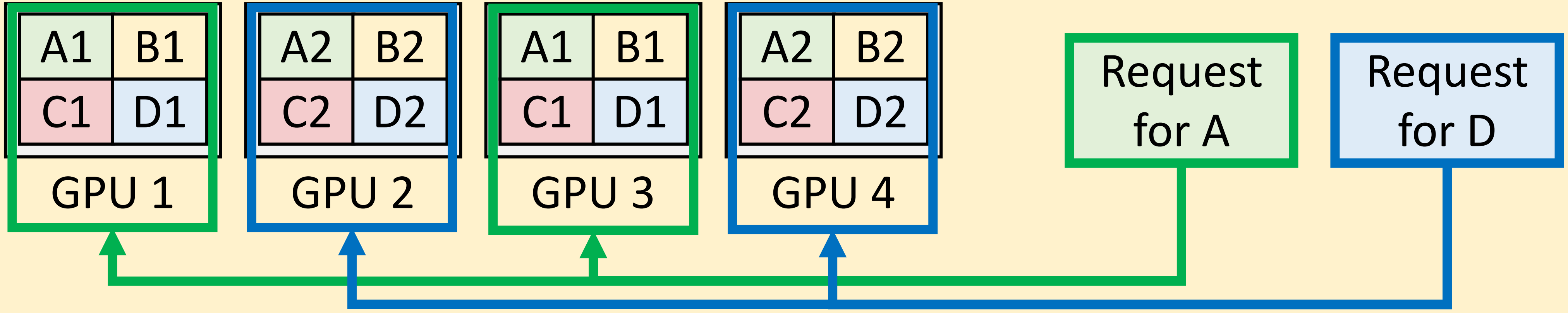
Existing GPU pooling is *not* effective

- *Not effective - limited number of models per GPU*
- Two existing approaches for GPU pooling: multiplexing and auto-scaling
- ***Multiplexing*** is static: serving with multiple (sharded) models per GPU



Existing GPU pooling is *not* effective

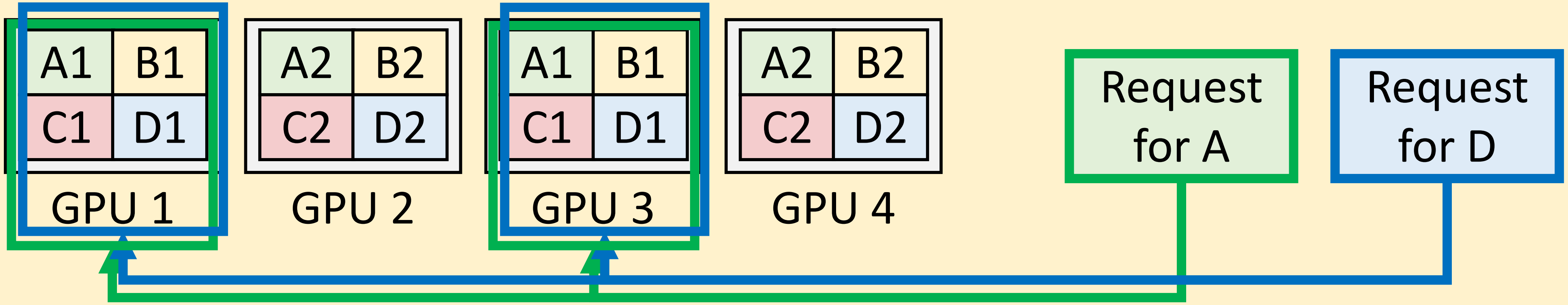
- *Not effective - limited number of models per GPU*
- Two existing approaches for GPU pooling: multiplexing and auto-scaling
- ***Multiplexing*** is static: serving with multiple (sharded) models per GPU



Temporal Multiplexing

Existing GPU pooling is *not* effective

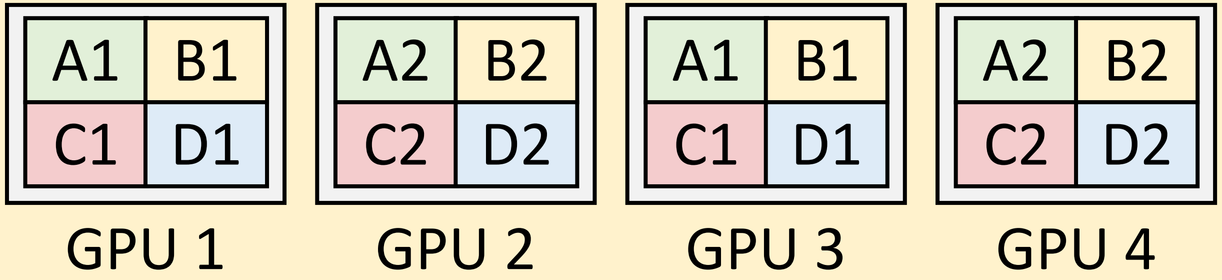
- *Not effective - limited number of models per GPU*
- Two existing approaches for GPU pooling: multiplexing and auto-scaling
- ***Multiplexing*** is static: serving with multiple (sharded) models per GPU



Spatial Multiplexing

Multiplexing is *limited*

- *Not effective - limited number of models per GPU*
- Two existing approaches for GPU pooling: multiplexing and auto-scaling
- ***Multiplexing*** is static: serving with multiple (sharded) models per GPU



8 models on 4 GPUs
=> 2 models per GPU

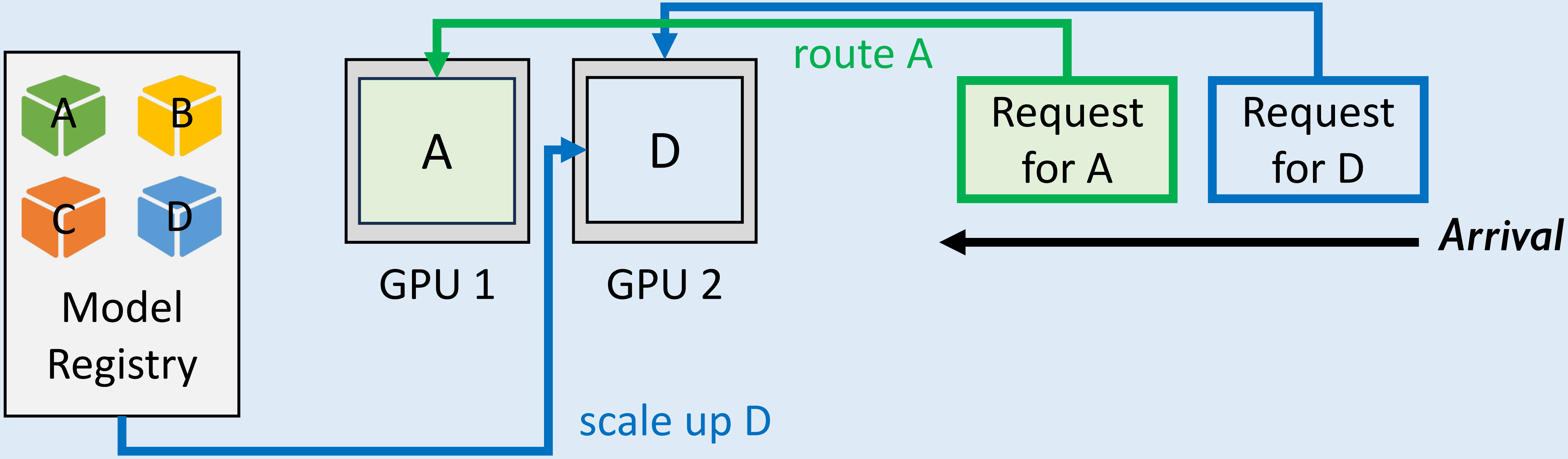
Limited by GPU memory capacity:
At most two 14B models (FP16) per 80GB GPU

Auto-scaling is more promising

❖ *Not effective - limited number of models per GPU*

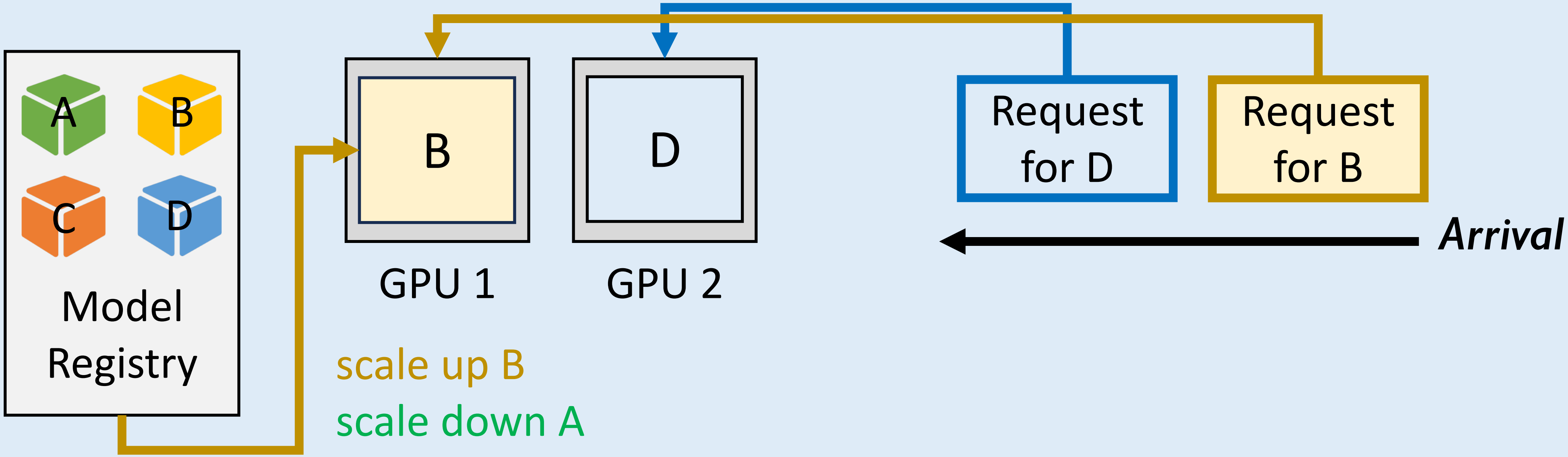
❖ Two existing approaches for GPU pooling: multiplexing and auto-scaling

❖ ***Auto-scaling*** is dynamic: changing model placement on GPUs over time



Auto-scaling is more promising

- *Not effective - limited number of models per GPU*
- Two existing approaches for GPU pooling: multiplexing and auto-scaling
- ***Auto-scaling*** is dynamic: changing model placement on GPUs over time

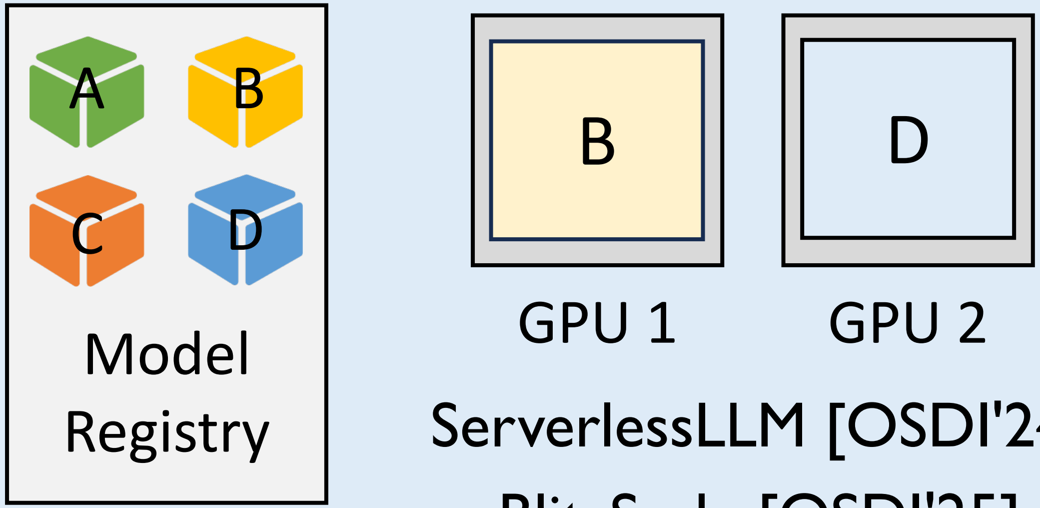


Auto-scaling is more promising

❖ *Not effective - limited number of models per GPU*

❖ Two existing approaches for GPU pooling: multiplexing and auto-scaling

❖ ***Auto-scaling*** is dynamic: changing model placement on GPUs over time



ServerlessLLM [OSDI'24]
BlitzScale [OSDI'25]

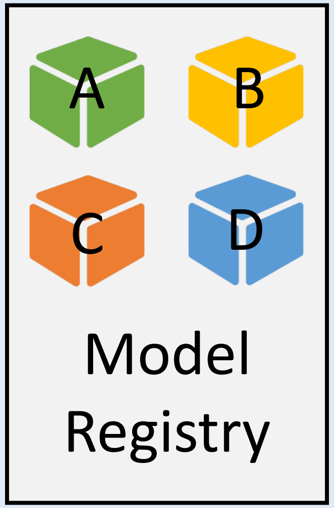
Not limited by memory capacity!

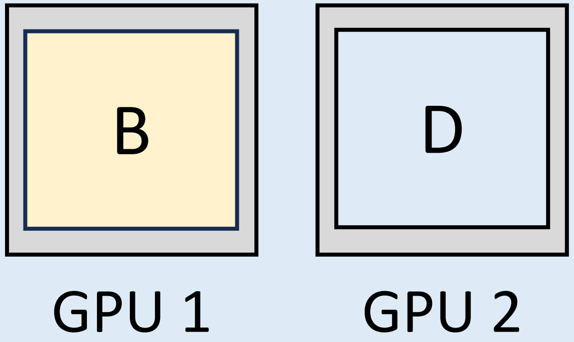
Auto-scaling is more promising

❖ *Not effective - limited number of models per GPU*

❖ Two existing approaches for GPU pooling: multiplexing and auto-scaling

❖ ***Auto-scaling*** is dynamic: changing model placement on GPUs over time



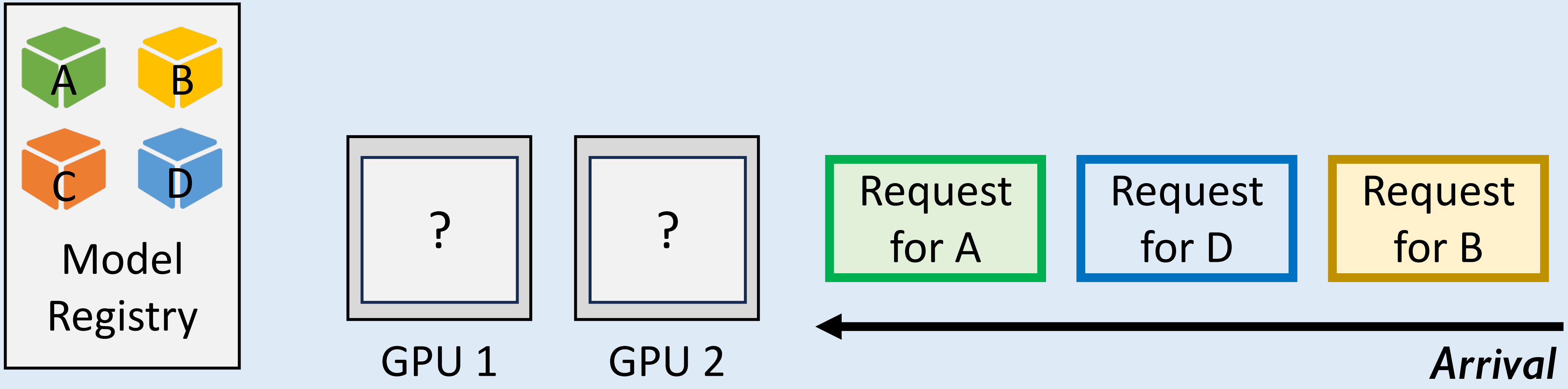


ServerlessLLM [OSDI'24]

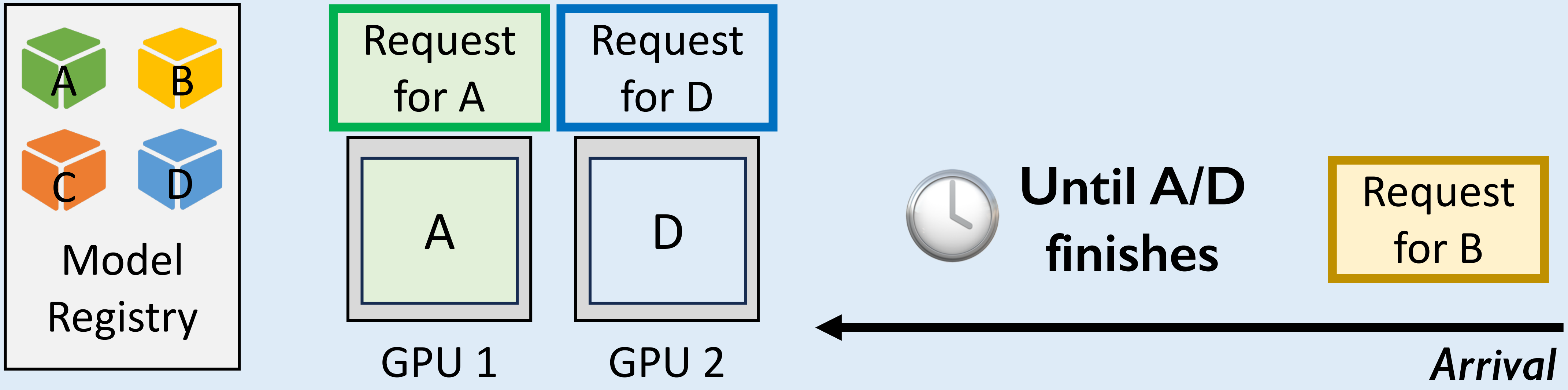
BlitzScale [OSDI'25]

Unlimited?

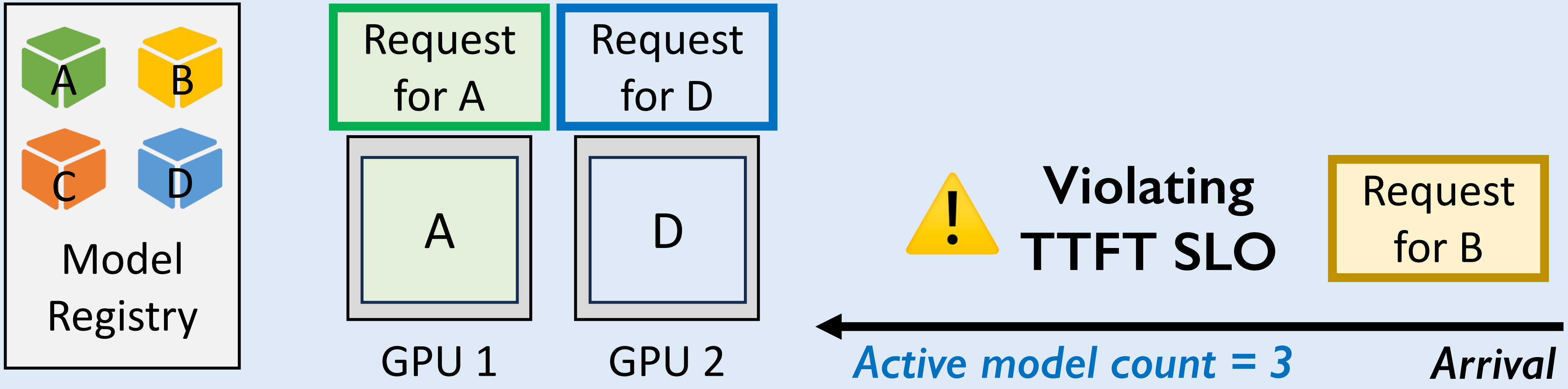
Existing auto-scaling is *still limited*



Existing auto-scaling is *still limited*



Existing auto-scaling is *still limited*



Limited by active model count:
Need more GPUs than #Model active at a time

Most models are active in a concurrent workload!

- Intuitively, because LLM requests tends to be very long.
- Formally,

Theorem 1. *Suppose the request arrival rate for each model follows a Poisson process with rate λ, and the average time to serve a request is T. The expected active model count $\mathbb{E}[m]$ is given by:*

$$\mathbb{E}[m] = M \cdot (1 - e^{-\lambda T}) \tag{1}$$

λ = 0.037 RPS
T = 16.79s
(Data @ Alibaba)

M/E[m] ≈ 2.16

Limited by active model count:
At most M/E[m] < 3 models per GPU

Most models are active in a concurrent workload!

- Intuitively, because LLM requests tends to be very long.
- Formally,

Theorem 1. *Suppose the request arrival rate for each model follows a Poisson process with rate λ, and the average time to serve a request is T. The expected active model count $\mathbb{E}[m]$ is given by:*

$$\mathbb{E}[m] = M \cdot (1 - e^{-\lambda T}) \quad (1)$$

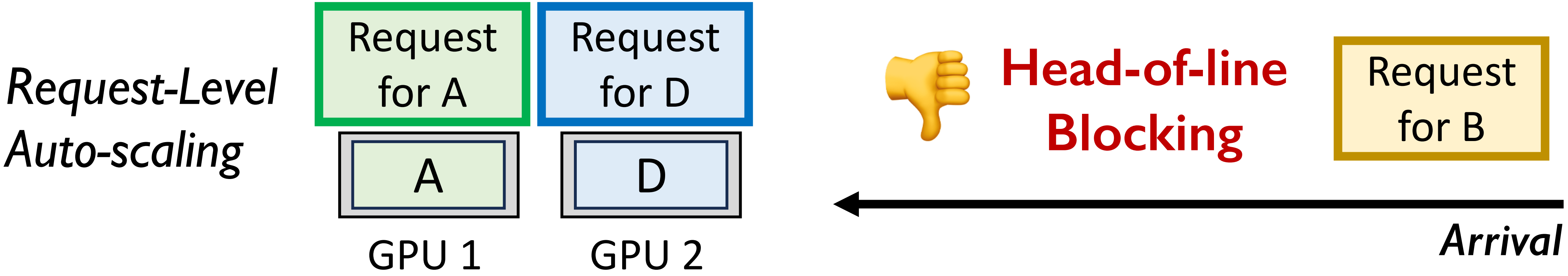
λ = 0.037 RPS
T = 16.79s

M/E[m] ≈ 2.16

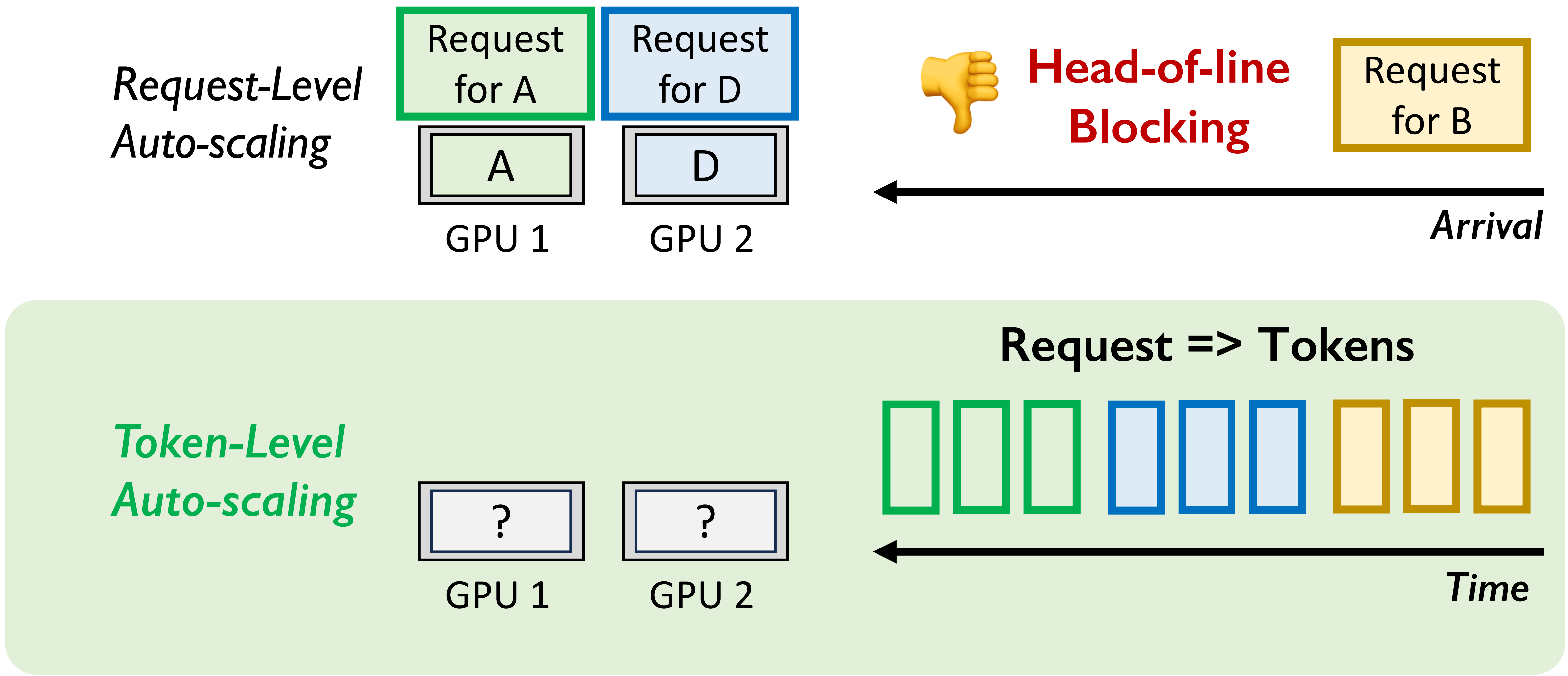
Still room for improvement!

👎 **Limited by active model count:**
At most M/E[m] < 3 models per GPU

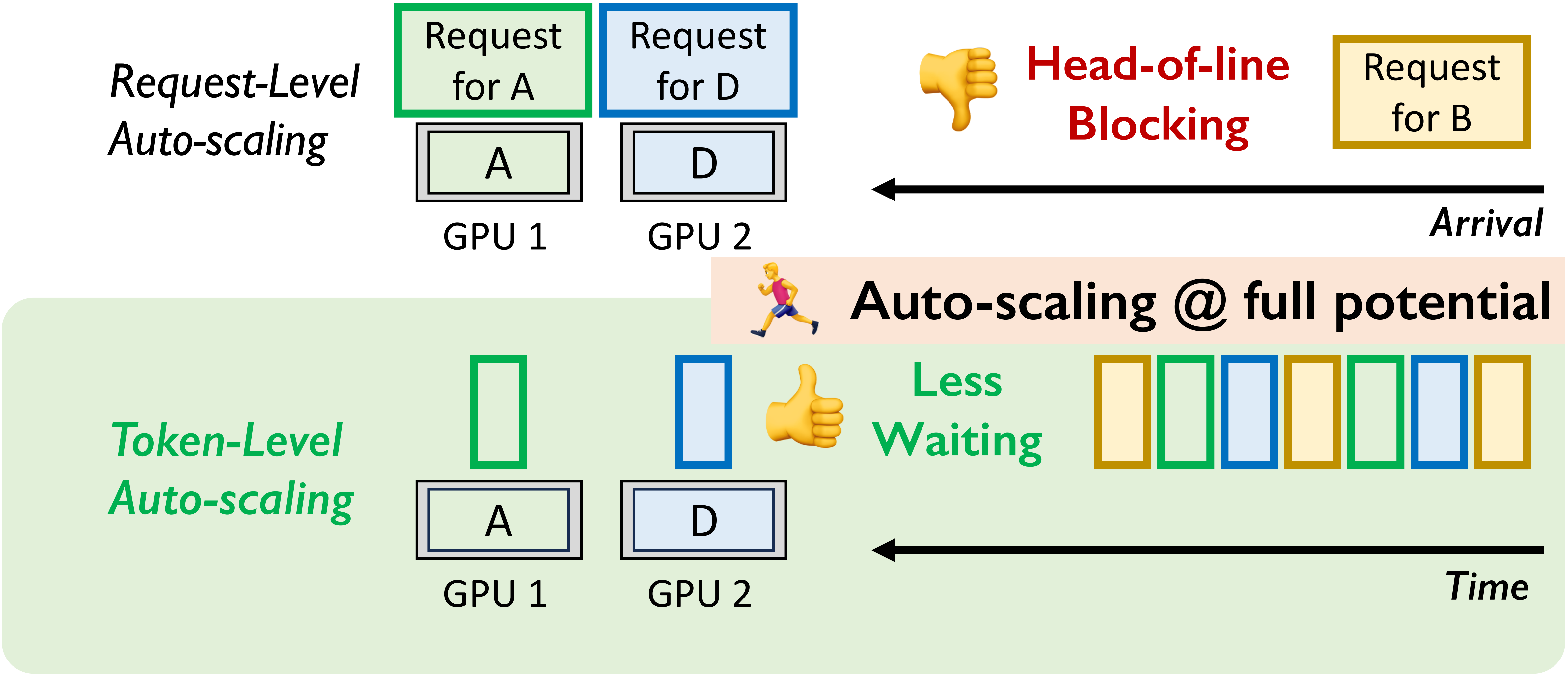
Key: Request-level auto-scaling is inadequate



Our approach: token-level auto-scaling



Our approach: token-level auto-scaling



Token-level auto-scaling: most effective

Concurrent LLM serving
(w/ GPU pooling)
- Multiple (n) models per GPU
- 0.2 x n RPS per GPU

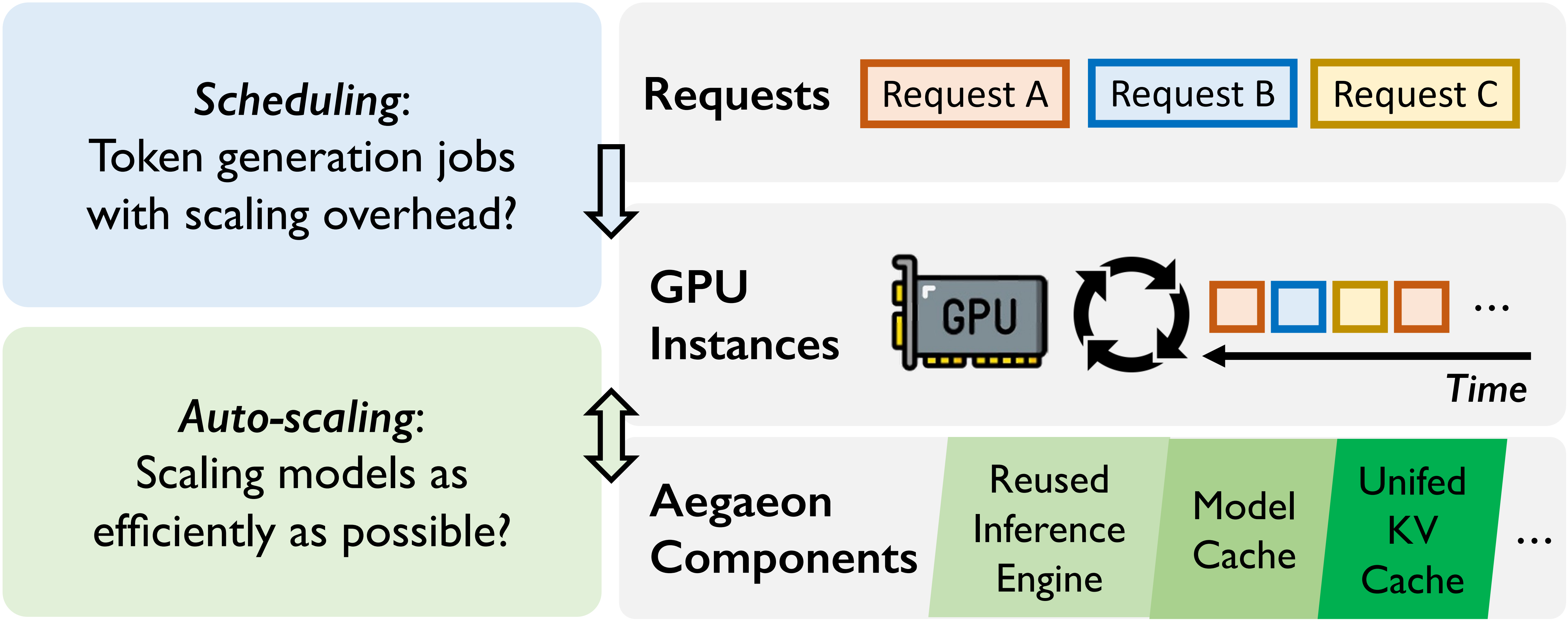
Effective: *Serve as many models as possible per GPU!*

Given M concurrent models (m active models):

	Multiplexing	Request-Level Auto-scaling	Token-Level Auto-scaling (ours)
#GPU Limitation	$M/2$ (memory)	m (active models)	$< m$
#Model/GPU (typical)	2	2~3	Up to 7 (later)

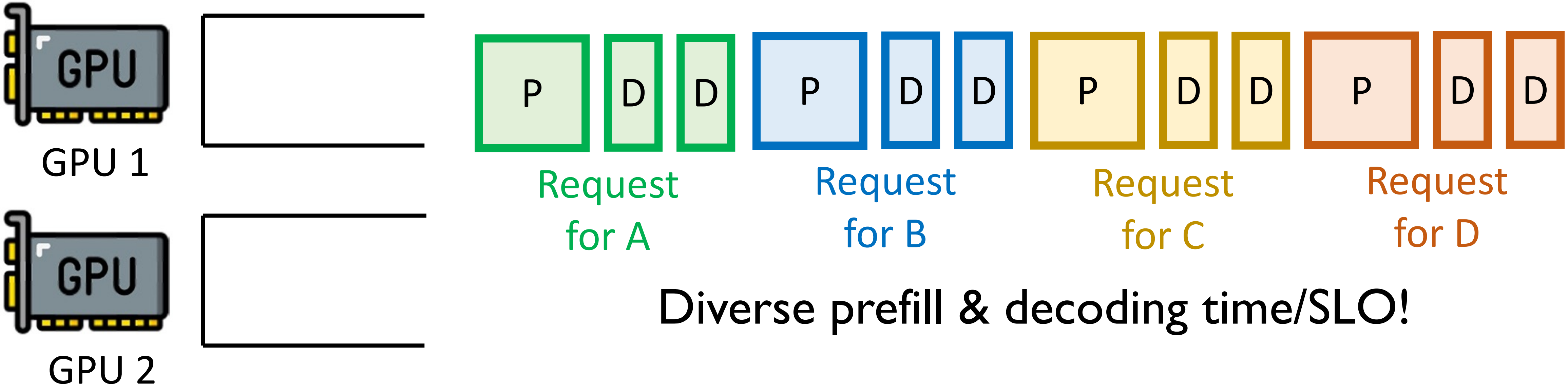
Design Overview

Aegaeon: token-level auto-scaling for concurrent LLM serving.



Token-level request scheduling

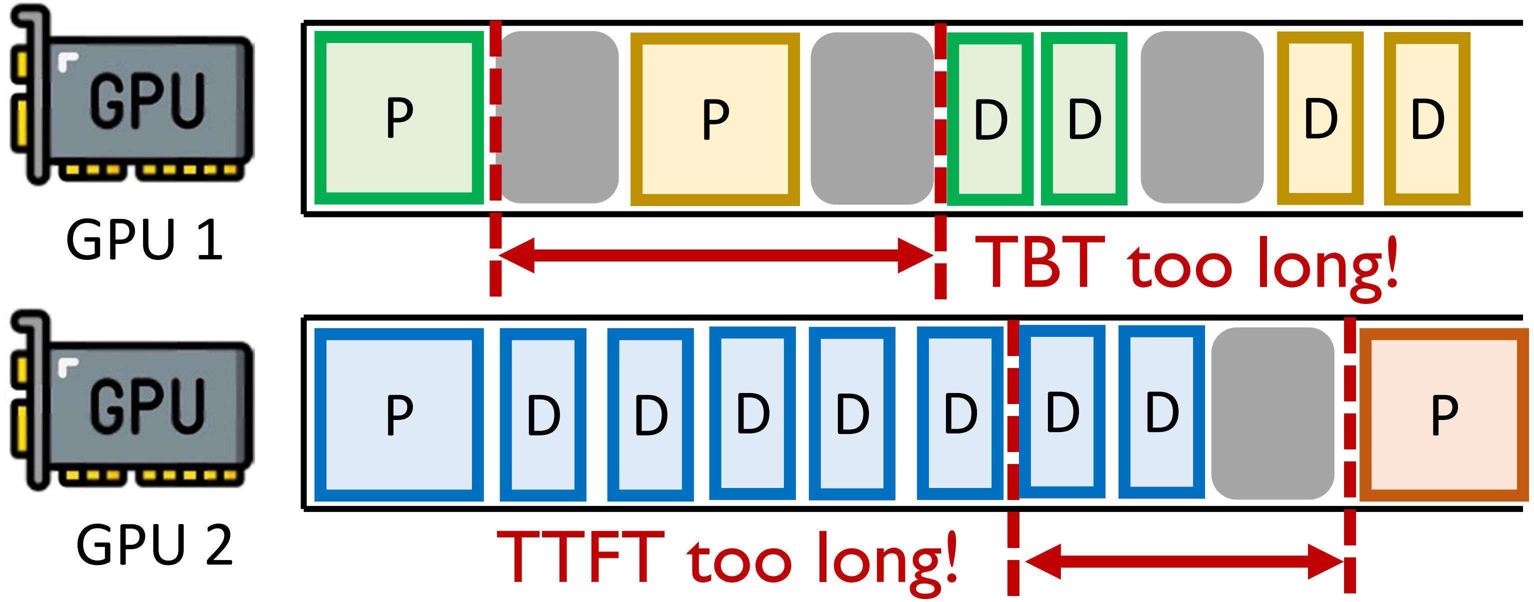
Goal: Given requests, #GPU, SLOs (TTFT/TBT),
maximize SLO attainment.



Diverse prefill & decoding time/SLO!

Scheduling with P/D Disaggregation

Goal: Given requests, #GPU, SLOs (TTFT/TBT), maximize SLO attainment.



Bursty prefill: violates TBT SLO

Long-tail decoding: violates TTFT SLO

Problem: Prefill-decoding interference.

Solution: Schedule with P/D disaggregation for simplifying scheduling design.

Prefill scheduling: grouped FCFS

❖ Prefill execution & auto-scaling overhead are comparable (seconds)

Goal: Avoid frequent auto-scaling during prefill.

Solution: Group prefill requests by models;
schedule groups with FCFS.

✅ Group: Reduce auto-scaling ⚖️ ✅ FCFS: Reduce starvation

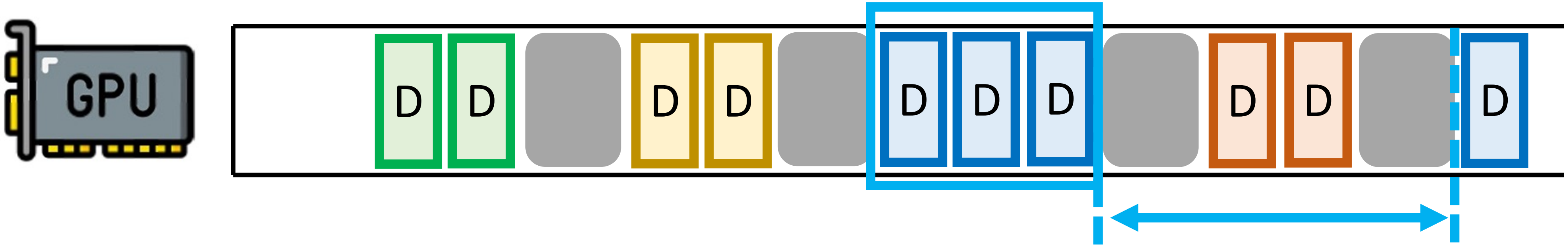
Decoding scheduling: batched Round-Robin

- Decoding execution (O(10) *ms*) often <u>shorter</u> than TBT SLO (100 *ms*) in industrial practice.
- Decoding tokens can be <u>streamed</u> to avoid user-perceivable stalls.
- After several decoding steps, a request earns some slack time that can be used to serve other models.

Goal: Systematically utilize the slack time during decoding to switch between models at the token-level.

Solution: Round-robin with calculated time quota and guaranteed SLOs.

Decoding scheduling: batched Round-Robin



No SLO violation:
masked by streaming previous tokens

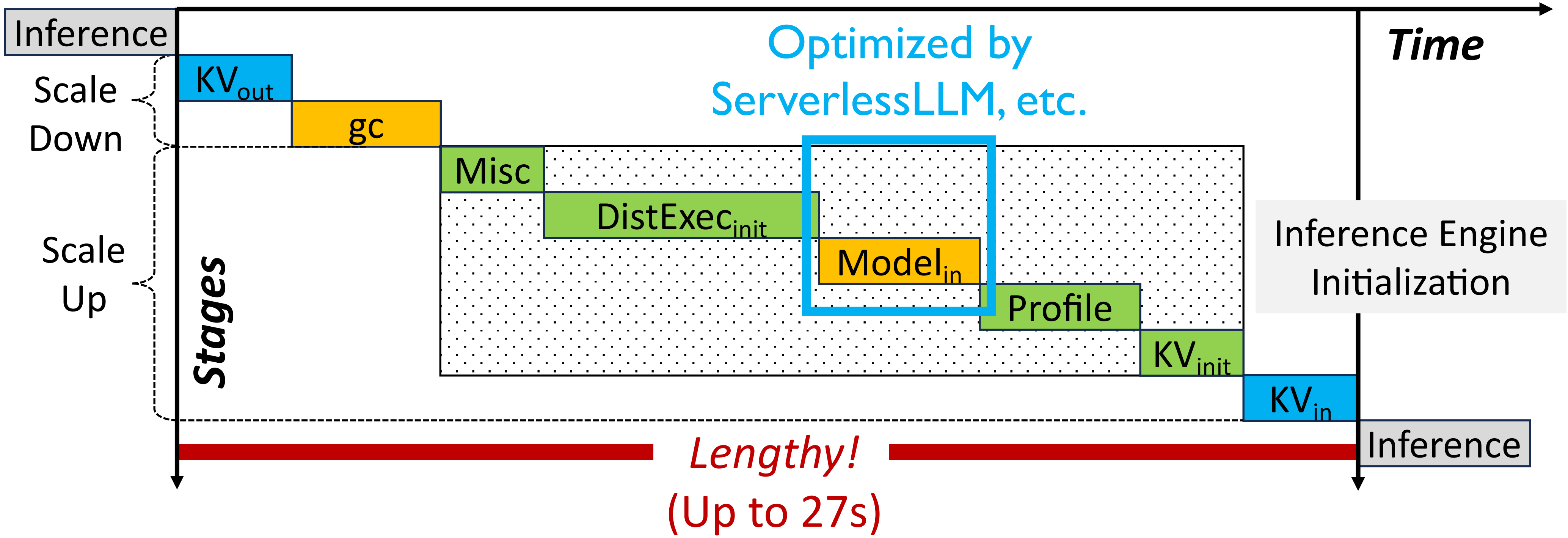
Detailed math in our paper!

$$q_i = \frac{c}{n_i \cdot (\alpha - \Sigma_k \frac{1}{n_k})}$$

$$\alpha = \max(\frac{c}{\min_k(n_k) \cdot \mathrm{Q_{MAX}}} + \Sigma_k \frac{1}{n_k}, 0.5)$$

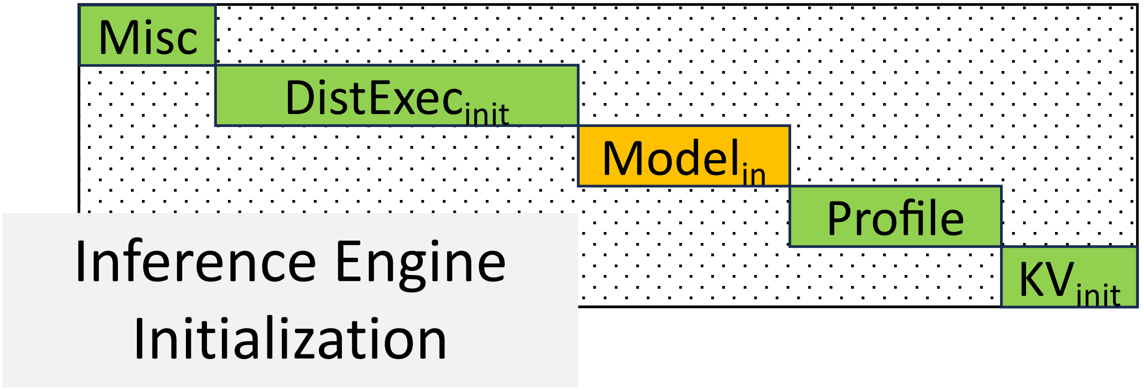
Auto-scaling: time-consuming by default

- Full sequence of preemptive auto-scaling:



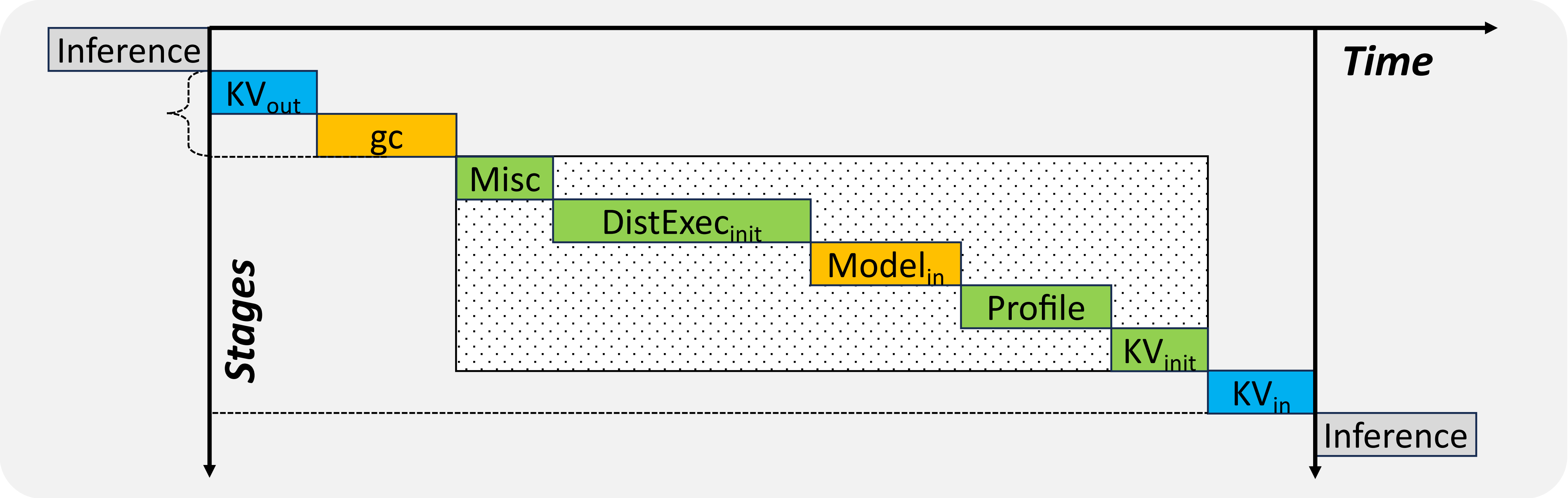
Goal: Conduct <u>full-stack</u> optimizations for accelerating auto-scaling.

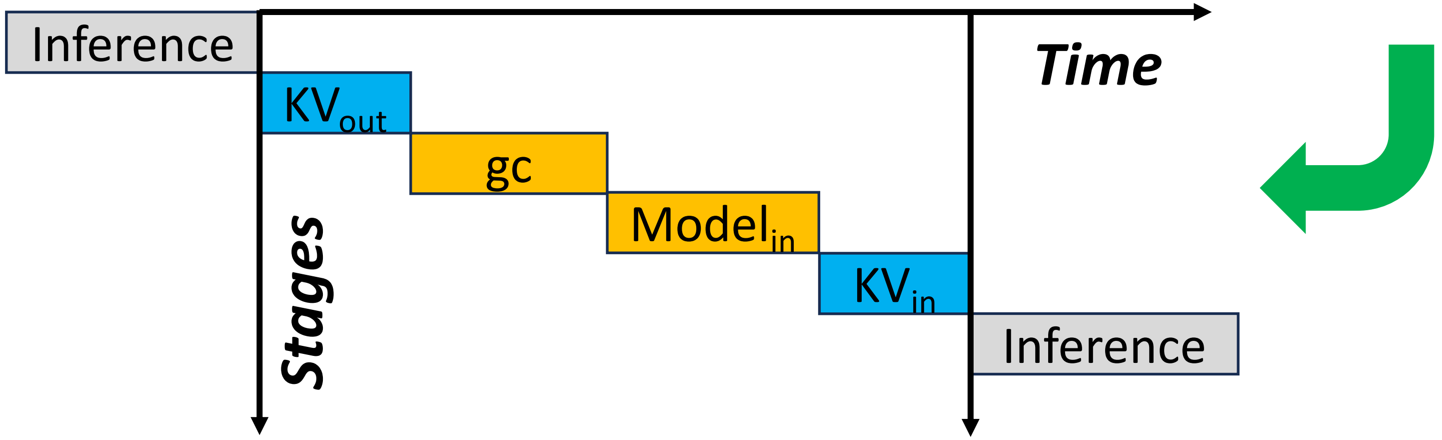
Auto-scaling w/ Component Reuse



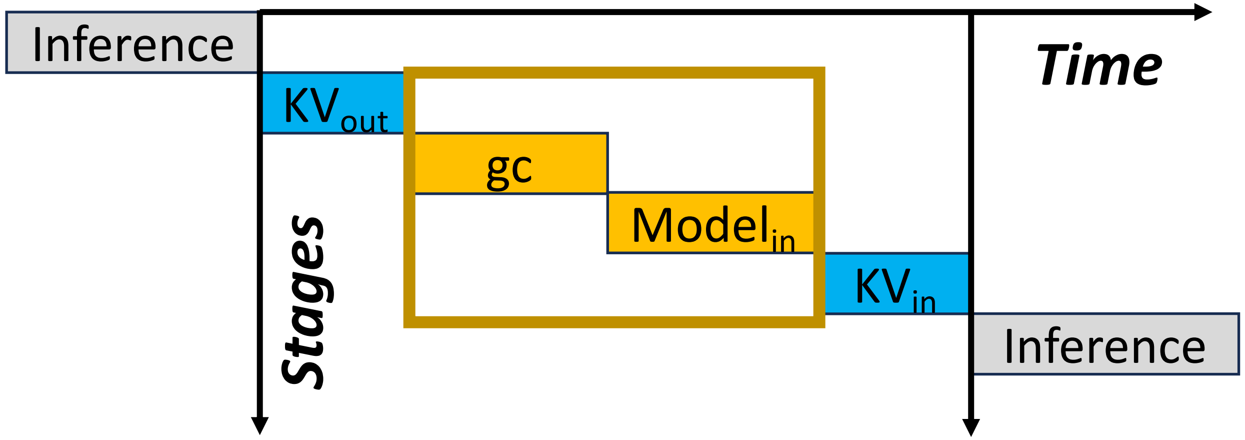
- Components (except the weights) can be shared across models
- **Components Reuse**: pre-build NCCL/Ray clusters, pre-profile, …

Auto-scaling w/ Component Reuse





Auto-scaling w/ Explicit Memory Management

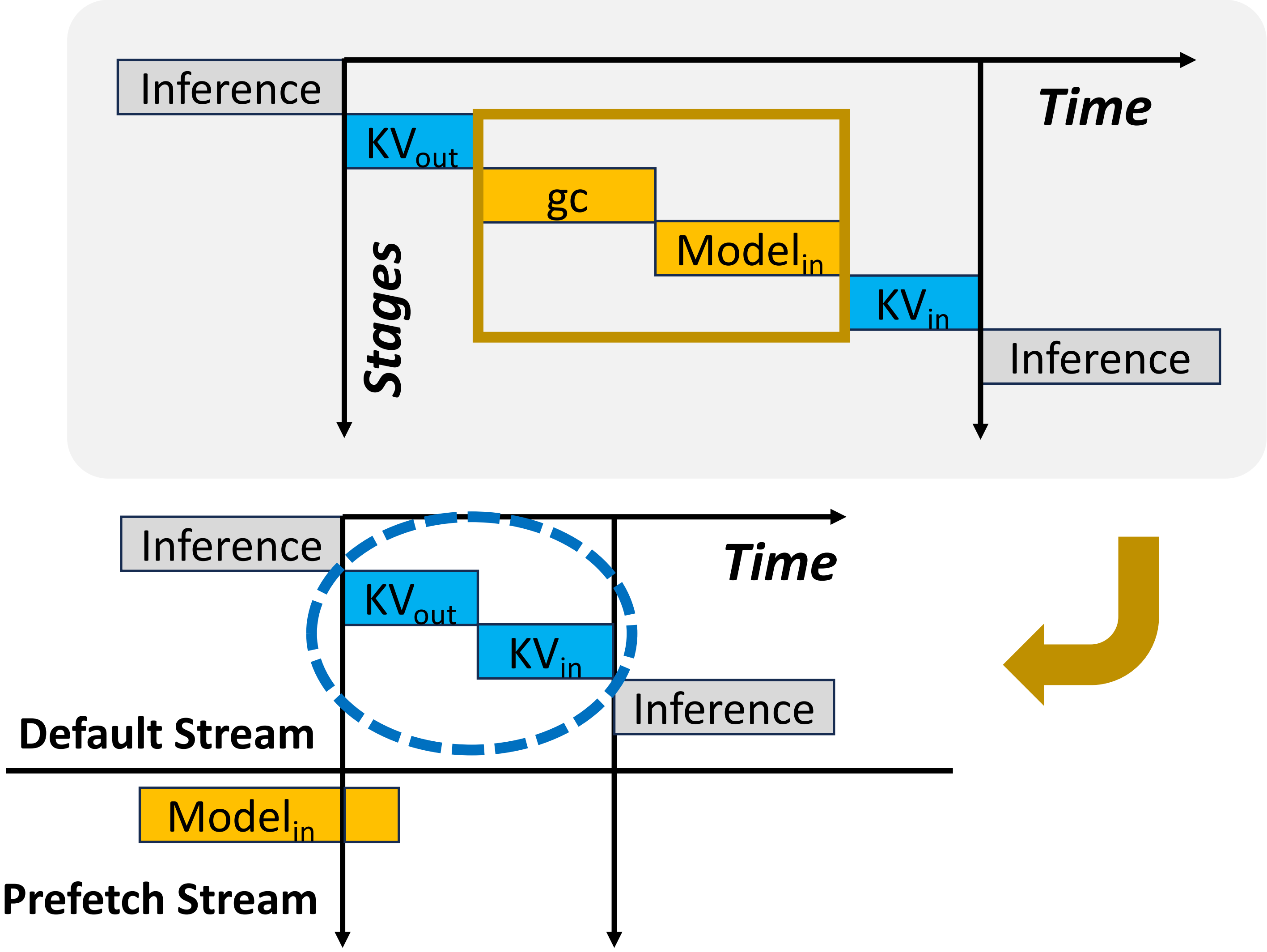


❖ Memory fragmentation during auto-scaling
- VRAM: Library (PyTorch) inefficiencies (`torch.cuda.empty_cache()`)
- DRAM: Different KV cache shapes

Solution: Explicitly manage VRAM and DRAM to avoid fragmentation.

✓ Custom VRAM allocator
✓ Unified KV cache: shape-specific on-demand allocation
✓ Model caching and prefetching

Auto-scaling: the final stretch

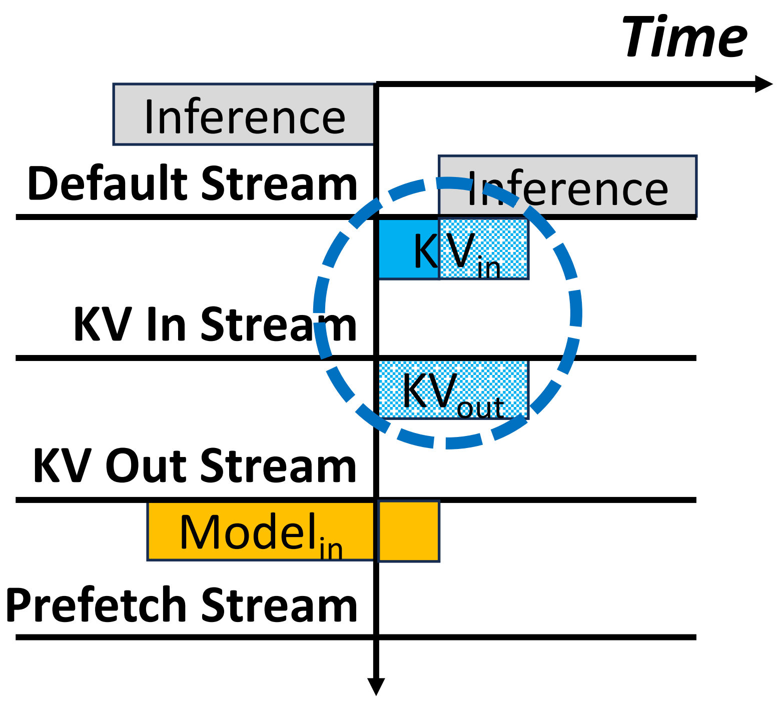


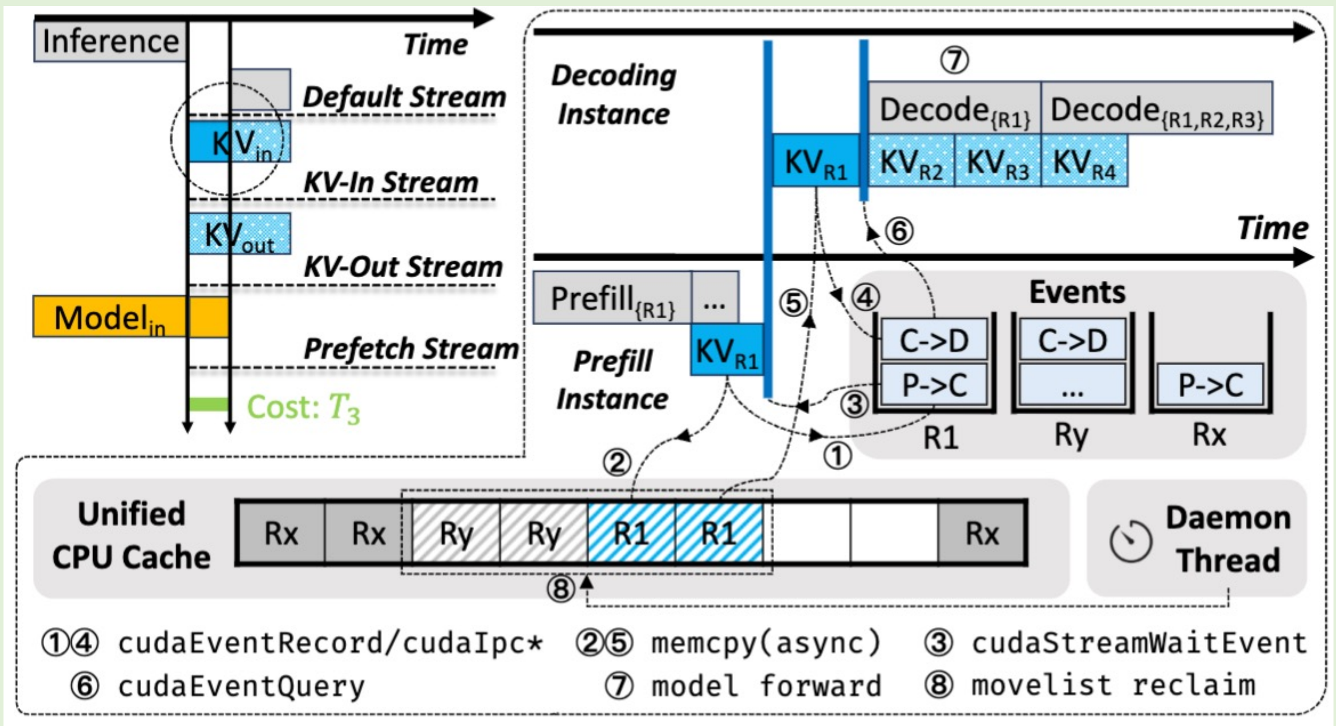
Goal: Maximally overlapping the KV cache transfers.

Auto-scaling w/ Fine-Grained KV$ Synchronization

- **Fine-grained KV cache synchronization** with <u>CUDA events</u>
  - Make transfers fully asynchronous
  - Then synchronize minimally with CUDA events

Full design in our paper!

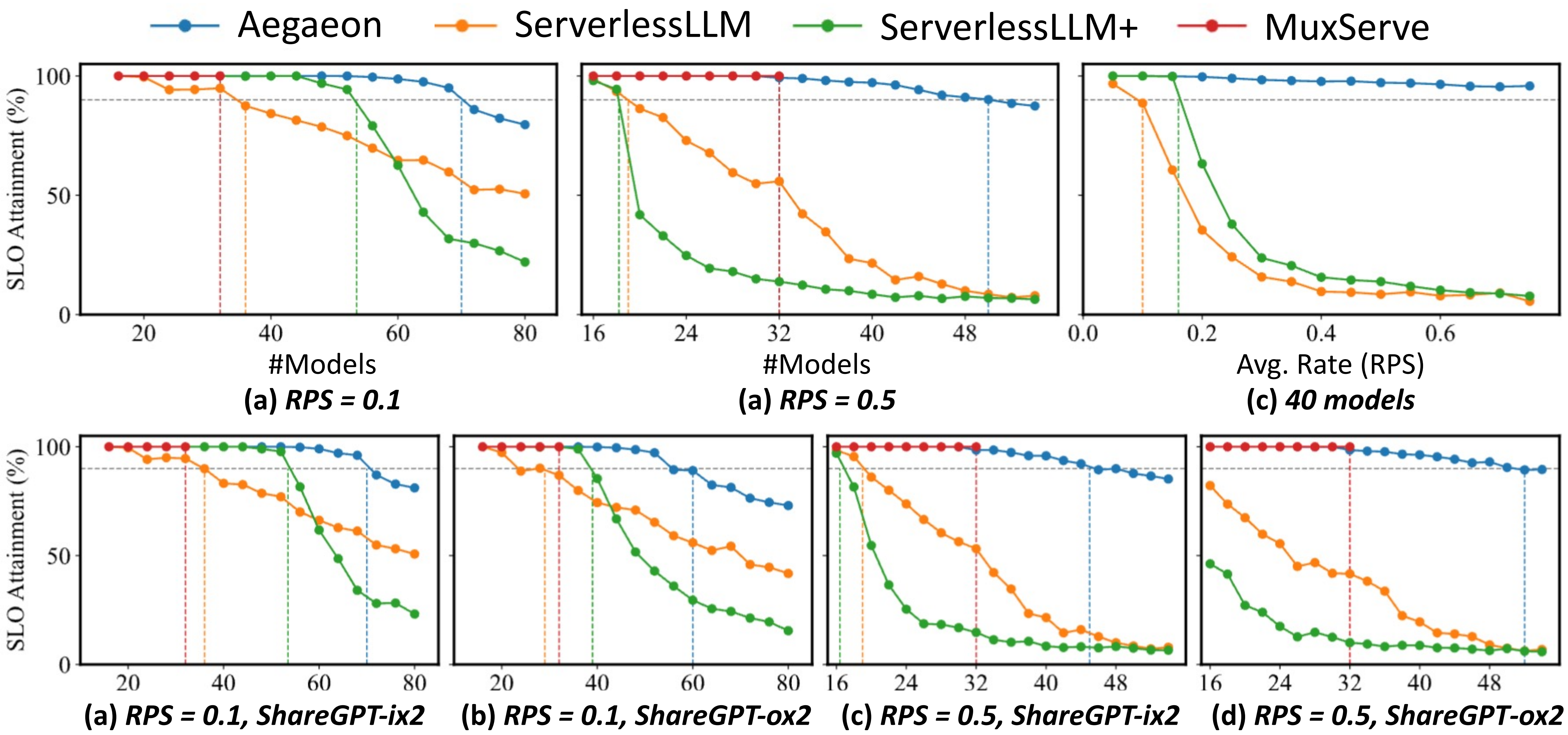




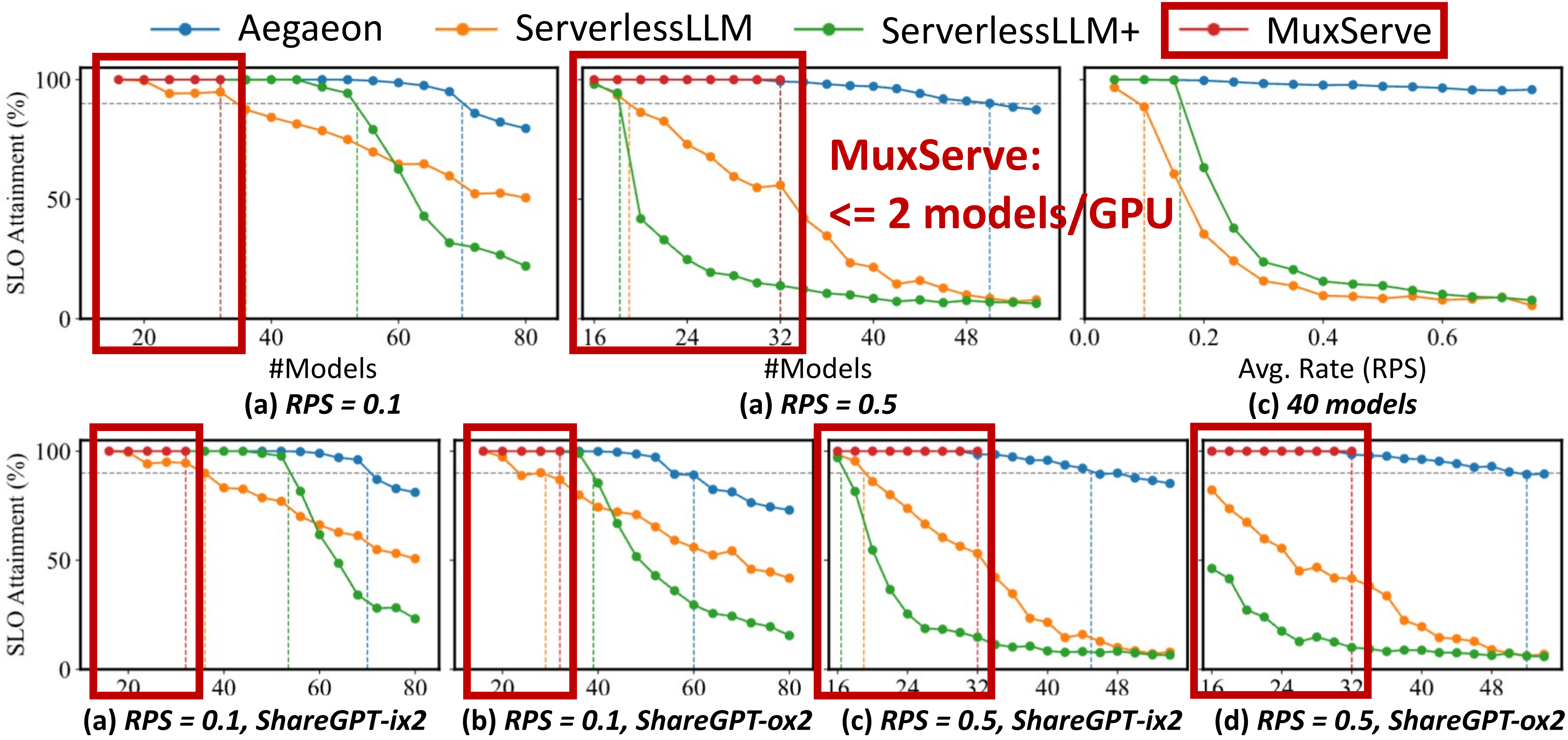
Evaluation Setup

- **Baselines**: ServerlessLLM (auto-scaling), MuxServe (multiplexing).
  - ServerlessLLM+: ServerlessLLM + SRTF scheduling (oracle)
- **Metrics**: SLO attainment. Base TTFT is 10s, TBT is 100ms.
  - Also scaled to 0.5x, 0.3x, and 0.2x.
- **Workload**: (1) ShareGPT + Poisson (2) real traffic for deployment
  - **Models**: 80 different models, including LLaMA, Qwen, InternLM, Yi, etc. The size ranges from 6B to 14B. 72B models are also evaluated.
- **Cluster**: 2x8 H800 nodes; 1TB host memory, PCIe 5.0.
  - **Deployment**: 6 GPUs for prefill, 10 for decoding.

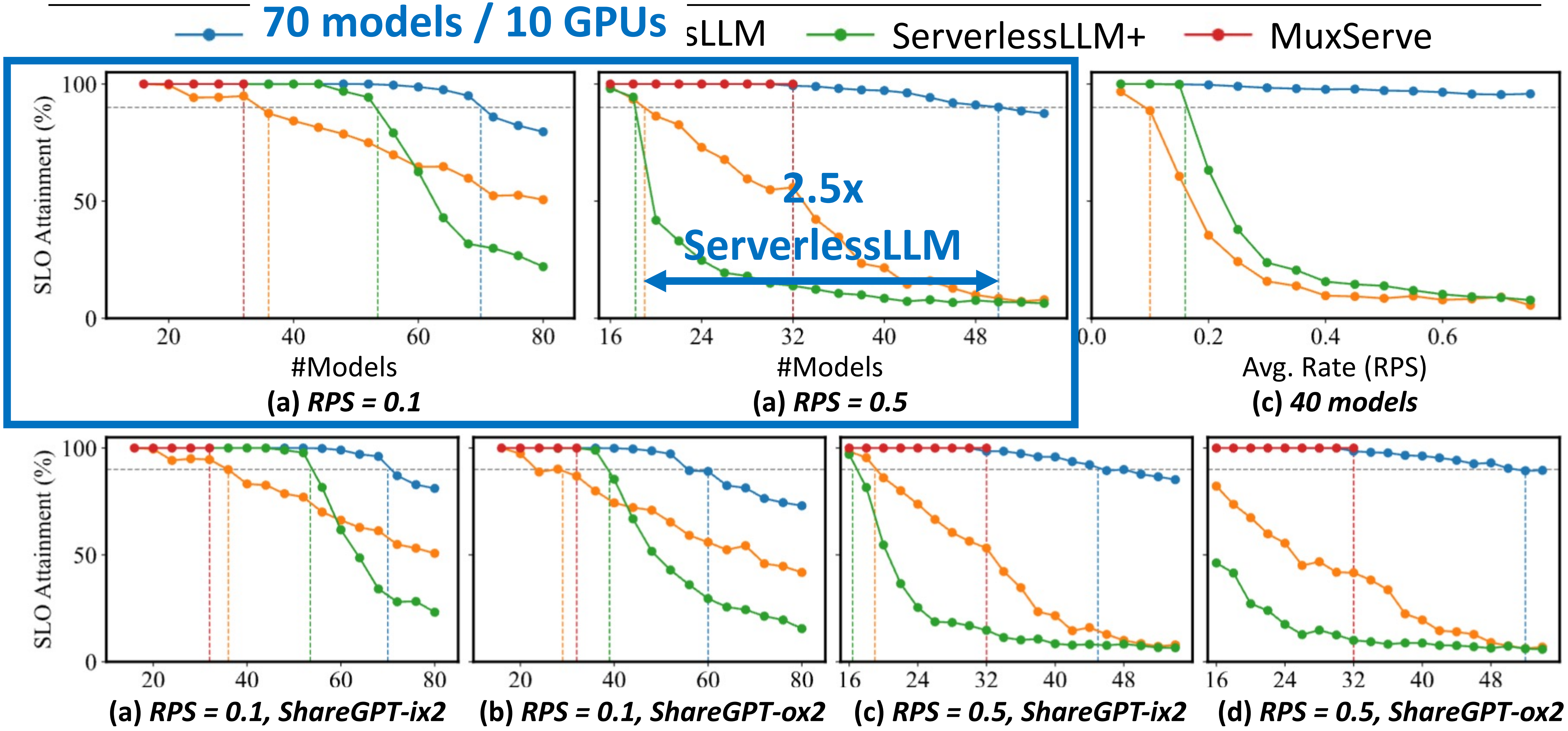
Evaluation: End-to-End Results



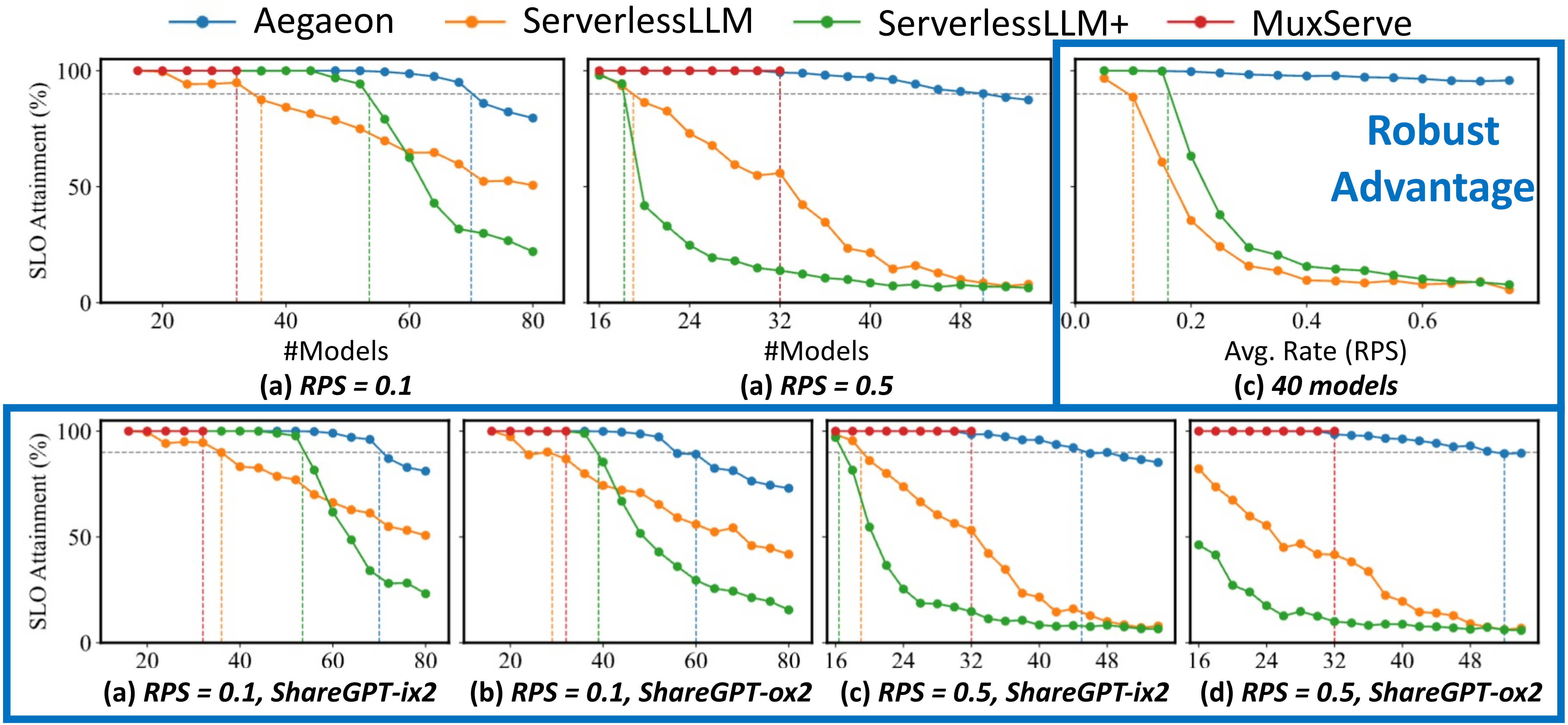
Evaluation: End-to-End Results



Evaluation: End-to-End Results

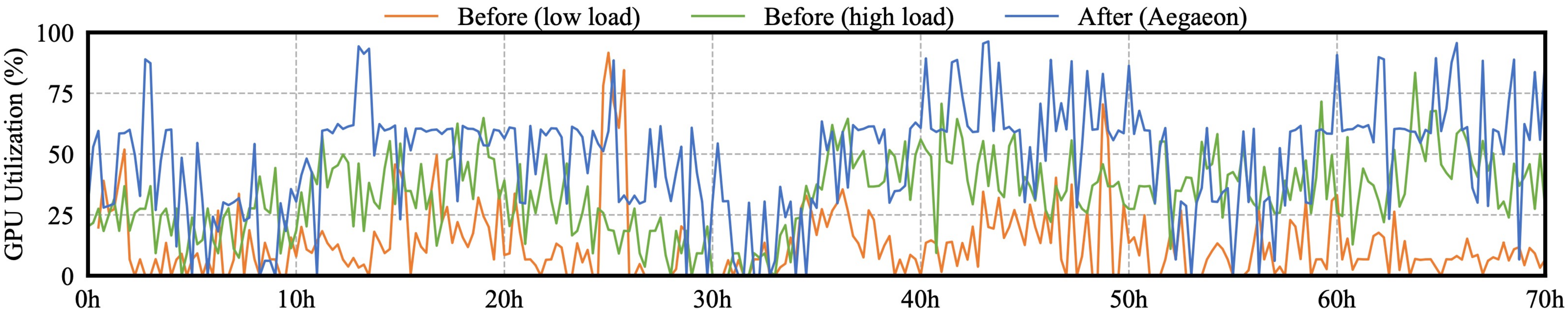


Evaluation: End-to-End Results



Evaluation: beta deployment

- *Beta-deployed* on a 213-GPU cluster (H20), serving nineteen 32-72B (TP=4), and twenty-eight 1.8B-7B models (TP=1).
- Previously served by 1,192 GPUs (*82% saving*).



Alibaba Cloud
Model Studio

Summary

Aegaeon proposes token-level auto-scaling to achieve effective GPU pooling for concurrent LLM serving:

- Workload modeling and effectiveness analysis *(as many models as possible!)*
- Token-level request scheduling algorithms
- Efficient auto-scaling optimizations

Aegaeon reduces GPU usage by up to 82% in beta deployment.

Thanks for Listening!

xiangyx@stu.pku.edu.cn

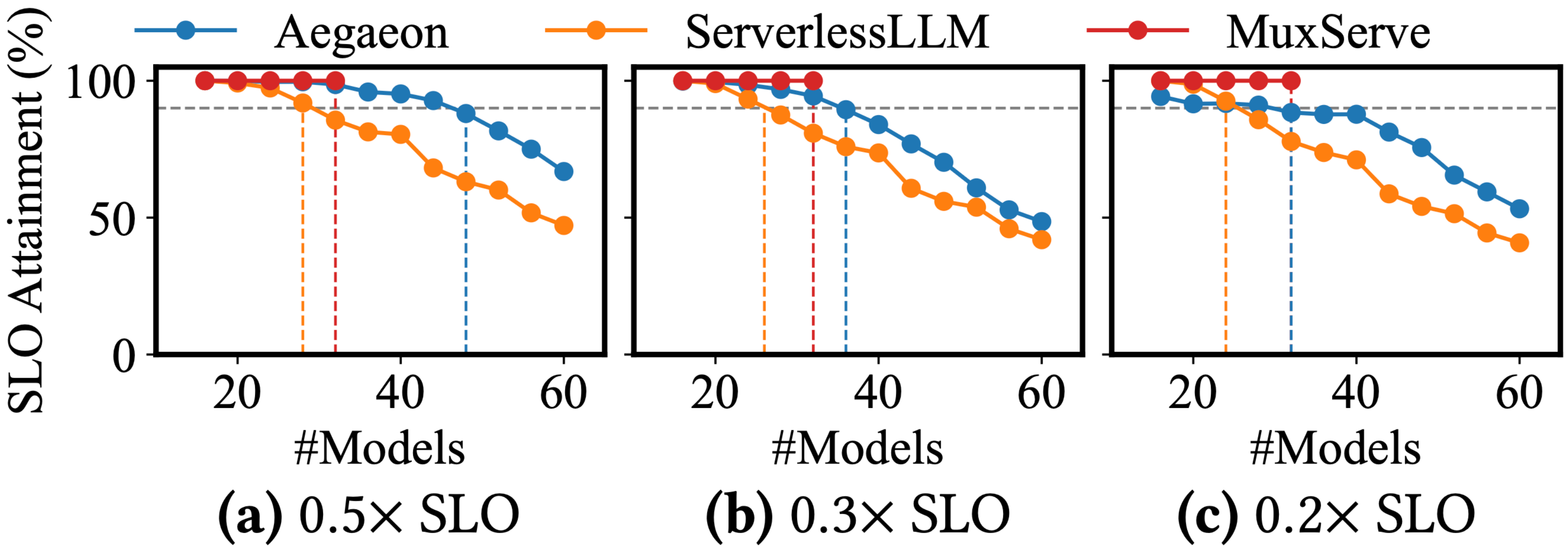
北京大學 PEKING UNIVERSITY

Alibaba Cloud

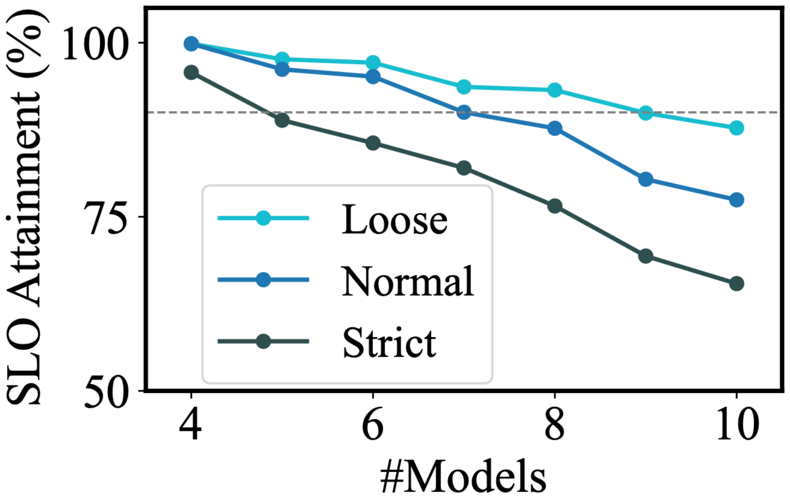
Paper

Evaluation: Alternative setups

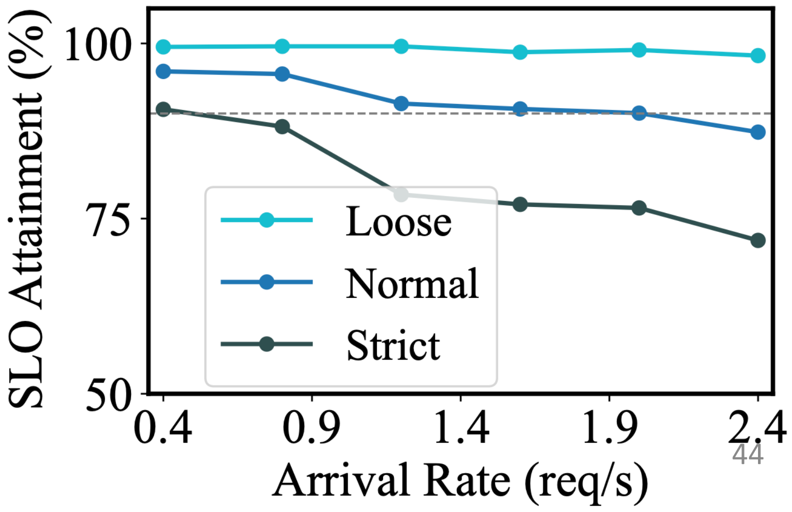
Tighter SLOs



4xA10



72B (TP4)



SGLang's DeepSeek Deployment Blog

LMSYS ORG

- Projects
- Blog
- About
- Donations
- Chatbot Arena (graduated)

Prefill

	DeepSeek Blog (excl. cache hit)	DeepSeek Profile	SGLang (Default)	SGLang + Simulated Perfect EPLB
Batch Size	N/A	16,384	16,384	16,384
Input Length	N/A	4,096	4,096	4,096
Throughput (per node)	32,206	62,713	50,302	59,337

Decoding

	DeepSeek Blog	DeepSeek Profile	SGLang (Default)	SGLang + Simulated MTP (Slow Attention)
Batch Size	N/A	128	256	128
KV Cache Length	4,989	4,096	2,000	4,000
Number of Nodes	18	16	9	9
Throughput (per node)	14,800	18,598	22,282	17,373